An Empirical Study of Knowledge Transfer in AI Pair Programming

Alisa Welter[1], Niklas Schneider[1], Tobias Dick[1], Kallistos Weis[1], Christof Tinnes[2], Marvin Wyrich[1], Sven Apel[1]
[1]Saarland University, Germany
[2]Siemens AG, Germany

***Abstract*—Knowledge transfer is fundamental to human collaboration and is therefore common in software engineering. Pair programming is a prominent instance. With the rise of AI coding assistants, developers now not only work with human partners but also, as some claim, with *AI pair programmers*. Although studies confirm knowledge transfer during human pair programming, its effectiveness with AI coding assistants remains uncertain.**

To analyze knowledge transfer in both human–human and human–AI settings, we conducted an empirical study where developer pairs solved a programming task without AI support, while a separate group of individual developers completed the same task using the AI coding assistant GitHub Copilot. We extended an existing knowledge transfer framework and employed a semi-automated evaluation pipeline to assess differences in knowledge transfer episodes across both settings. We found a similar frequency of successful knowledge transfer episodes and overlapping topical categories across both settings. Two of our key findings are that developers tend to accept GitHub Copilot's suggestions with less scrutiny than those from human pair programming partners, but also that GitHub Copilot can subtly remind developers of important code details they might otherwise overlook.

***Index Terms*—AI pair programming, knowledge transfer**

I. Introduction

Pair programming has been shown to be an efficient technique, yielding higher quality and more readable code compared to individual efforts [1, 2]. While pair programming can reduce errors, its lower time efficiency [1] often leads to it being underused in practice. Meanwhile, advances in AI have begun to reshape coding practices. The advent of GitHub Copilot [3] in late 2021, followed by other coding assistants, has noticeably changed how code is written [4, 5]. GitHub, for example, reports a growing number of companies adopting GitHub Copilot[1] to increase productivity, claiming that one in three Fortune 500 companies now employs it.[2] Similarly, Google's CEO Sundar Pichai notes that over a quarter of new code at Google is now AI-generated.[3]

This emerging technical landscape has sparked the idea of "AI pair programmers", that is, replacing one of the two human developers in a pair programming setting with an AI coding assistant. A key question, however, is whether an AI coding assistant can effectively replace the second human while preserving the core benefits of traditional pair programming. In particular, one significant long-term benefit of pair programming is knowledge transfer [6], which is the focus of our research.

Knowledge transfer frequently happens during pair programming, as developers continuously discuss issues and their solutions as they progress [2, 6, 7]. This creates learning opportunities through knowledge exchange and valuable discussions that contribute to solving tasks or resolving challenges. For instance, a developer might explain some domain logic, guide their peer through a complex algorithm, or simply demonstrate a new keyboard shortcut in their coding environment. It is important to note that knowledge transfer can occur subtly and does not always result from direct questions and answers. It can emerge naturally during collaboration, such as when developers proactively share insights or demonstrate best practices.

In the literature, different views on knowledge transfer in pair programming have been explored [6, 8]. In general, collaborative techniques such as discussions and peer teaching have been shown to deepen learners' understanding of concepts and problems compared to working alone [9, 10]. With the idea of AI pair programming, a second human is not always involved in such pair activities. Kuttal et al. [7] studied a digital agent with a simulated face that engaged programmers through questions and suggestions, but its limited reasoning hindered effective knowledge exchange. Imai [11] investigated changes in productivity and code quality in pair programming with GitHub Copilot compared to a human partner. In their study, GitHub Copilot enhanced productivity but reduced code quality, as indicated by an increase in deleted lines.

The broader picture is that, while existing research has either focused on understanding and improving human–human pair programming or on using agents to enhance efficiency, only little attention has been paid to directly comparing knowledge transfer between human–human pair programming and pair programming with AI coding assistent. We therefore set out to investigate knowledge transfer in programming with AI coding assistants, particularly GitHub Copilot, and compare it to traditional pair programming. To measure knowledge transfer, we devise a knowledge transfer framework based on the works of Zieris and Prechelt [6] and Kuttal et al. [7]. Based on this framework, we establish a semi-automated pipeline to evaluate the data collected from our empirical study, in which a total of 19 developers either code as pairs or use GitHub Copilot on a programming problem that required participants to develop algorithms and integrate them into a cohesive project framework.

[1] https://github.com/features/copilot/
[2] https://www.microsoft.com/investor/reports/ar24/index.html
[3] https://blog.google/inside-google/message-ceo/alphabet-earnings-q3-2024/

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00022

Our results show that knowledge transfer occurs in both human–human pair programming and human–AI pair programming. GitHub Copilot can even subtly provide helpful reminders, such as suggesting missing database commits. While human pairs engage in more frequent but sometimes distracting exchanges, GitHub Copilot promotes more focused interactions—though users tend to trust its suggestions more readily, raising questions about critical engagement.

In summary, our contributions are as follows:

- We unify and extend the perspectives of Zieris and Prechelt [6] and Kuttal et al. [7] into a single framework for knowledge transfer in the context of AI coding assistants.
- We conduct an empirical study comparing knowledge transfer in AI-assisted and traditional pair programming, revealing key differences between the two settings in the context of a coherent and non-trivial programming task.
- Drawing on our findings, we discuss how combining both approaches can enhance development efficiency and focus while preserving the benefits of human collaboration.

II. Background and Related Work

This section reviews prior work on knowledge transfer, pair programming, and AI assistants, forming the conceptual basis of our study alongside the framework in Section III.

A. Knowledge Transfer

Knowledge transfer plays a crucial role, for example, in education [12, 13, 14, 9, 15]. Numerous studies indicate that peers working together often outperform individuals on certain tasks through discussions, explanations, and the sharing of ideas [13, 14]. However, understanding the nature of these processes requires a profound analysis. According to constructivist learning theory, knowledge cannot be transmitted directly as it is not a tangible entity [16, 12]. Instead, individuals must explore and construct knowledge through experiences and by resolving conflicts with their existing knowledge [17]. In general, knowledge transfer involves actions and mechanisms that facilitate this construction process, such as explanations and discussions that can take place in pair programming.

B. Pair Programming

Pair programming is a technique in which two developers work collaboratively at a single computer [18]. We use this technique as a baseline for our study, as it has been researched extensively [19, 20, 1]. Zieris and Prechelt [6] explore knowledge transfer in pair programming and propose a framework detailing how knowledge transfer processes are structured. They identify factors that facilitate effective knowledge transfer [6, 21, 22] with the overarching goal of providing practical advice for programmers. Dybå et al.'s meta-analysis [1] examines the effectiveness of pair programming by reviewing the results of 15 studies focusing on various aspects such as work speed, effort, and quality: With pair programming, tasks are completed faster and software quality improves. However, pair programming is more expensive than individual programming due to the increased number of person-hours required. Additionally, the benefit of pair programming declines with programmer expertise and task complexity [1].

C. AI Coding Assistants

While pair programming is traditionally between two humans, an AI coding assistant can take on tasks typically handled by a human partner. GitHub Copilot is a popular example, powered by the large language model (LLM) Codex [3]. Released by Microsoft in 2021, it was one of the first AI assistants to be integrated into Integrated Development Environments (IDEs) via plugins. At the time of this study, only Codex was available as an underlying model, although more recent versions of GitHub Copilot now support various LLMs. In this study, we differentiate between *human–human pair programming*, where two developers work together on one computer, and *human–AI pair programming*, where a developer collaborates with an AI coding assistant, GitHub Copilot.

Several studies have focused on the performance of GitHub Copilot regarding different aspects, mostly correctness. Nguyen and Nadi [23] use LeetCode[4] questions as prompts for GitHub Copilot to solve programming tasks in four different languages. Using LeetCode tests and code complexity metrics, they find that correctness depends strongly on the programming language, with Java performing best, and GitHub Copilot's code being highly comprehensible as indicated by low complexity values. Peng et al. [24] observe an improvement in the time needed to solve programming tasks when using AI assistants in a controlled experiment.

Concerning the analysis of pair programming in a setting where one of the partners is replaced with a digital agent, Kuttal et al. [7] explore an approach where the agent, represented by a simulated face on the screen, interacts with the human programmer by prompting questions and offering ideas. They find no significant difference in the typical advantages of pair programming, such as productivity, quality, and self-efficacy, maintaining the benefits of the traditional approach. Participants trust and accept the simulated face on the screen as a partner, yet the agent struggles to explain logic or contribute original ideas, impairing effective knowledge exchange. Kuttal et al. conducted a Wizard-of-Oz study, with a human operator generating the suggestions and controlling the computer agent behind the scenes. In contrast, our study examines an actual tool that can autonomously generate code.

In a study conducted by Imai [11], participants are asked to implement a Minesweeper game in Python through pair programming, with GitHub Copilot partially serving as a programming partner. The objective of the study is to measure productivity and code quality. The study measures productivity by lines added and quality by deletions. GitHub Copilot users are most productive but produce lower-quality code.

Ma et al. [25] compare human–human pair programming and human–AI pair programming across effectiveness metrics (learning, cost, collaboration, communication) and call for more human–AI pair programming research to fully leverage AI in

[4] https://leetcode.com/

pair programming. Barke et al. [26] perform a study to identify two key interaction modes with GITHUB COPILOT: In the acceleration mode, programmers have a clear plan and use GITHUB COPILOT as an autocomplete tool, quickly accepting suggestions without disrupting their workflow. In exploration mode, programmers are unsure how to solve a problem and use GITHUB COPILOT to explore various approaches [26].

Stray et al. [27] interview developers and report that AI assistants were said to improve efficiency and reduce stress during individual programming, giving developers more time to focus on solving complex tasks. However, the also noted that AI assistants were rarely used in human–human pair programming.

Fan et al. [28] conduct a quasi-experiment with students and find that human–AI pair programming improves motivation, reduced anxiety, and outperforms individual work, while human–human pair programming leads to the highest sense of collaboration and social presence [28].

To summarize, previous research focused on understanding and improving human–human pair programming or using agents to boost efficiency. Only little attention has been given to directly comparing knowledge transfer between human–human pair programming and human–AI pair programming, which is however imperative to understand the impact of the increasing use of AI as a programmer. Our study is the first to investigate *knowledge transfer* in pair programming with a *real* AI tool, used in the context of a *coherent and non-trivial programming task*. The task goes beyond simple line-level code completion and requires participants to comprehend documentation and understand the broader program structure. We see this as a timely and necessary step toward understanding how AI is reshaping the role of the human programmer in a rapidly evolving area.

III. A KNOWLEDGE TRANSFER FRAMEWORK FOR HUMAN–AI PAIR PROGRAMMING

To compare knowledge transfer in human–human pair programming and human–AI pair programming, we devise a knowledge transfer framework based on Zieris and Prechelt [6] and Kuttal et al. [7], ensuring comparability between the two settings. We analyze several metrics, as well as the content, and success of knowledge transfer episodes, as described below. To define knowledge transfer, we first define a knowledge gap.

Definition 1 (Knowledge gap). A *Knowledge gap* is a disparity between the information or understanding a person currently possesses and the knowledge a pair member considers relevant for their software development context including their current task and the software project it is embedded in [22].

Intuitively, a knowledge gap can only be bridged through the assimilation of new information, thereby leading to our definition of knowledge transfer.

Definition 2 (Knowledge transfer). Any attempt to close a perceived knowledge gap by exchanging existing knowledge or building new knowledge is considered *knowledge transfer* [22].

The key aspects of knowledge transfer according to Zieris and Prechelt [6] in the context of pair programming are also applicable for human–AI pair programming. Zieris and Prechelt [6] focus on the structure of knowledge transfer, with which we will start.

In our framework, conversations are made of utterances. Since measuring knowledge transfer in individual utterances is impractical, we group related utterances into episodes.

Definition 3 (Episode). An *episode* is defined as a sequence of related utterances focused on a single topic, initiated either by a person explicitly expressing a need for knowledge or a person sharing knowledge they possess with others.

Since an episode is initiated based on a perceived knowledge gap rather than an objectively existing one, a new episode may occur even when no actual knowledge gap is present.

To allow statistical characterization of episodes, we define the *length* and *depth* of an episode. Intuitively, the length of an episode can be measured by counting the utterances it contains.

Definition 4 (Length of an episode). The *length of an episode* is the number of utterances that belong to it, including the utterances belonging to episodes layered within it.

Episodes can be layered if a new topic is introduced before the current episode is complete. This layering requires a return to the topic of the initial episode. To formalize this structure, we define the concept of depth within episodes.

Definition 5 (Depth of an episode). The *depth of an episode* refers to the total number of episodes that contain the given episode, with the top-level episode having a depth of one.

In other words, the depth of an episode refers to its nesting level within other episodes. A top-level episode, not nested within any other episode, is assigned a depth of one. Each time an episode is nested within another, depth increases by one.

Consider the following example: A developer is using a new API and asks their partner for help, initiating an episode focused on API usage. While following their partner's explanation, a missing dependency causes an error, prompting a nested episode on installing it. Once the dependency issue is resolved, the discussion returns to the initial topic of using the API. In this example, the length of the episode on API usage includes all utterances, including those from the inner dependency episode, while the inner episode only contains its own utterances. The depth of the inner episode is 2, as it is enclosed within the episode on API usage, which has a depth of 1.

Finish Types. Zieris and Prechelt [6] define attributes to characterize knowledge transfer episodes. We focus on how an episode ends—its *finish type* (Table I).

Based on the four concrete finish types that Zieris and Prechelt introduce (i.e., TRANSFERRED, GAVE UP, LOST SIGHT, and UNNECESSARY), we define five finish types of episodes in our framework. In particular, we split the finish type TRANSFERRED into two finer-grained finish types: ASSIMILATION and TRUST. To define these finish types, we first define the concept of a *customer*.

Definition 6 (Customer). The person who is in need of knowledge is called *customer* [6].

Table I: Definition of knowledge transfer finish types, based on Zieris and Prechelt [6]

Finish Type	Definition	Example
ASSIMILATION	The customer fully understands and internalizes the knowledge.	*"I get it"* or rephrasing the explanation in their own words.
TRUST	The customer accepts the contribution without full understanding, relying on the partner's expertise.	*"Fine with me"* or *"If Copilot says so, it'll be right."*
GAVE UP	The attempt to close the knowledge gap is abandoned due to frustration or complexity.	*"This is too complicated, never mind."*
LOST SIGHT	The episode ends because the pair gets sidetracked, and the original goal is no longer pursued.	*[Starts something unrelated while leaving the question unanswered]*
UNNECESSARY	The knowledge is deemed irrelevant or redundant.	*"I don't think this is important right now."*

Using Definition 6, we define the finish types in our framework. We start with the two finish types corresponding to TRANSFERRED, both of which describe situations in which the customer receives knowledge.

Definition 7 (*ASSIMILATION*). The episode concludes with the customer fully understanding and internalizing the knowledge shared by the partner. Knowledge transfer is complete and effective.

ASSIMILATION would include saying something like "I get it" because the customer understands the reasoning and agrees based on their own context. Instead of resulting in a short affirmative answer, transferred knowledge can also be rephrased by the customer.

Definition 8 (*TRUST*). The episode ends with the customer accepting the partner's contribution without fully understanding it, relying on the partner's perceived expertise or correctness. Knowledge is transferred but not deeply comprehended.

An example of the TRUST finish type would be the statement "Fine with me", which reflects acceptance of the information based on confidence in the source, rather than through personal understanding or critical engagement. Since genuine understanding is essential for applying knowledge effectively, we distinguish TRUST from ASSIMILATION even though such episodes can also be considered to represent a successful outcome.

However, knowledge has still been received even if not processed. The next three finish types are concerned with scenarios where knowledge transfer is not completed successfully.

Definition 9 (*GAVE UP*). The episode ends due to the customer or pair abandoning the effort to bridge the knowledge gap, often from frustration, complexity, or lack of progress. No knowledge transfer occurs.

In this case, a customer might express frustration with phrases like "This is too complicated, never mind", signaling disengagement from the knowledge exchange. This outcome is undesirable, as it results in no meaningful transfer, leaving the knowledge gap unresolved.

Definition 10 (*LOST SIGHT*). The episode terminates because the pair diverges from the original knowledge transfer goal, becoming sidetracked or overwhelmed, leaving the knowledge gap unresolved.

LOST SIGHT reflects a situation in which either the customer or their partner gets sidetracked. This can result in the start of a new episode, leaving the current knowledge transfer unfinished and the knowledge gap unresolved.

Definition 11 (*UNNECESSARY*). The episode ends when the pair recognizes the knowledge in question is irrelevant or redundant to the task, terminating knowledge transfer as it is deemed non-essential.

For instance, a customer might say "I don't think this is important right now," reflecting a situation in which the current knowledge transfer is no longer the priority. In such cases, the knowledge gap persists because the focus has shifted away from the original objective.

Table II: Definition of knowledge transfer topic types, based on Kuttal et al. [7]

Topic type	Definition	Example
TOOL	Knowledge about the IDE or how to use the tool.	*"There's a key bind for going back a tab"*
PROGRAM	Knowledge about the programming language itself or it's syntax.	*"I could also just aggregate the chars. I think they work like strings in that way."*
BUG	An error in the code.	*"I think we are missing code in that function."*
CODE	The code itself; i.e., what they are programming	*"So I'm gonna need to use a for loop as well."*
DOMAIN	The task (in case of [7]: implement tic-tac-toe game)	*"The goal is to get three of ... marker ... in a row."*
TECHNIQUE	About the techniques being used (i.e., test driven development, pair programming)	*"So if it's test driven development, I want to start by writing a test, I think."*

Topic Types. Zieris and Prechelt [6] propose that every utterance should be labeled with an information type. The overall information type of an episode is then derived from the most frequently occurring type among its individual utterances. Regarding the information type, Zieris and Prechelt [6] do not suggest a concrete classification scheme. In contrast, Kuttal et al. [7] explicitly address the content aspect of knowledge transfer. During their studies, they identified six distinct topics frequently discussed by developers (see Table II).

Instead of using the information type attribute defined by Zieris and Prechelt [6], we thus classify utterances using topic types based on the classes established by Kuttal et al. [7]. Although this list of classes cannot be exhaustive, it provides a structured overview of common topics discussed in knowledge transfer episodes. Specifically, the episode's topic type is determined by the most frequently occurring topic type among

its constituent utterances. This approach aligns with the models proposed by both Zieris and Prechelt [6] and Kuttal et al. [7], as both sets of categories can apply consistently at the utterances level, ensuring compatibility between the two models.

In summary, our knowledge transfer framework facilitates a comprehensive analysis of knowledge transfer for both our human–human pair programming and human–AI pair programming settings.

IV. Methodology

In our study, we examine differences in knowledge transfer between human–human pair programming and human–AI pair programming. We chose GitHub Copilot because it is the most widely used coding assistant integrated into IDEs. In what follows, we outline our research goals, study setup, and data acquisition framework, along with details on data collection and processing procedures.

A. Research Questions

We first investigate the extent of conversations and interactions that facilitate knowledge transfer in human–human pair programming and compare these results to the human–AI pair programming setting to determine whether programmers using the AI assistant experience similar knowledge transfer effects as in human–human pair programming.

RQ 1: To what extend do the frequency, length, and depth of knowledge transfer episodes differ between human–human pair programming and human–AI pair programming?

We also examine how the two settings differ and align content-wise. We analyze topic type frequencies, identify the most discussed topic type, and assess whether this differs between settings. The goal is to understand for which topic type GitHub Copilot can transfer knowledge to developers and where it may fall short.

RQ 2: How do the quality and diversity of knowledge transfer episodes, including topic types and finish types, vary between human–human pair programming and human–AI pair programming?

B. Study Design

We conduct a controlled experiment with two groups—a human–human pair programming group and a human–AI pair programming group—and both groups complete the same programming task in Python. The human–human pair programming group consists of pairs working together without GitHub Copilot, while the human–AI pair programming group includes individuals using GitHub Copilot. Participants are randomly assigned to one of the two groups.

Participants. Before the main study, we conducted a trial run to ensure the task was feasible and the technical setup worked as intended. We recruited students in the late stages of their bachelor's programs or the early stages of their master's programs. All participants were either teaching assistants for foundational computer science courses or active members of the student council, ensuring a basic level of academic engagement and communication skills. In total, 19 students took part in the study, resulting in 6 human–human pair programming sessions and 7 human–AI pair programming sessions.

Pre-test questionnaire. The study begins with a pre-test questionnaire, where participants self-assess their general programming skills and Python expertise along several dimensions using rating scales from 1 to 10.The questionnaire includes a personal skill assessment relative to their peers [29], their experience with GitHub Copilot, their familiarity with pair programming, and their overall programming experience.

Participants then receive standardized verbal instructions covering project structure, time limit, and task location. Participants in the human–human pair programming group are instructed to discuss with their partners, while those in the human–AI pair programming group, having no human partner, are asked to think aloud, providing a spoken record of their thoughts for evaluation.

Setting. Each participant works on a computer with an IDE set up, where the project is opened and necessary packages installed. For the human–AI pair programming group, an account is configured within the IDE to access GitHub Copilot, and the plugin is activated. They are shown an introductory video on using GitHub Copilot, while the human–human pair programming group receives an explanation of pair programming. Both groups may use all resources, except AI assistants for the human–human pair programming group. All participants may ask questions before the session begins.

From this point, screen activity and conversations (or think-aloud processes) are recorded until submission or time expiration, resulting in audio and video files for evaluation. Participants work on the task for 45 minutes, with no expectation to complete it. This aspect is communicated to participants to avoid time pressure.

Programming problem. Participants worked on a programming problem where certain features needed to be implemented within an existing codebase of approximately 400 lines including both Python code and comments, distributed across 5 files. This setup ensures that participants have to develop an understanding of the broader context while working within a manageable scope. Specifically, participants received the source code for a password manager—a console application managing multiple user accounts, each with their own password entries. After logging in, users can create, modify, view, and delete entries, which include a name, description, and password. The password manager stores data in a database, which is the focus of participants' implementation tasks, while the database schema and terminal interaction code are already provided.

Task. Participants' task is to complete functions marked with a #TODO comment, each with a provided signature and a doc string explaining the task. These functions are confined to a single file and focus on database access (using SQLAlchemy[5]) and building strings for console output. Two tasks involve

[5] https://www.sqlalchemy.org/

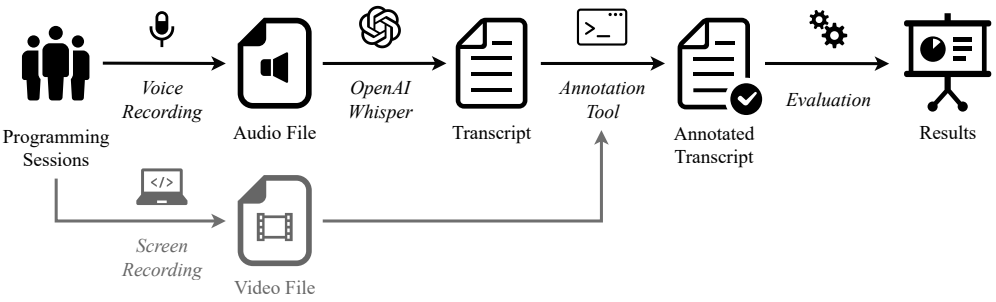

Figure 1: Overview of our data processing pipeline. While the annotated transcripts are primarily derived from voice recordings, screen recordings were consulted as needed to provide additional context.

string manipulation (e.g., building a formatted table of entries), while the remaining six focus on database interactions. Often, it is necessary to check conditions before modifying data (e.g., verifying if an object with a given name or id exists). The task was chosen such that, at the time of the study, GITHUB COPILOT could provide meaningful assistance but not solve the task outright with a single completion. Furthermore, we designed the task in a way that requires participants to understand documentation, integration logic, and the overall program structure. Hence, the setup approximates a real-world scenario while remaining feasible within the study constraints.

C. Operationalization and Data Processing

We evaluate knowledge transfer using our framework introduced in Section III to compare human–AI pair programming and human–human pair programming.

To compare knowledge transfer, we analyze the distribution of quantities such as the number, depth, and length of episodes (see Section III), which help us understand knowledge transfer dynamics. Each new episode reflects a perceived knowledge gap, with knowledge transfer defined as efforts to close that gap by exchanging or building new knowledge. To address RQ2, we examine the topic type (e.g., TOOL, PROGRAM, BUG, CODE, ...) and finish type (e.g., ASSIMILATION, GAVE UP, TRUST, ...) of these episodes.

We collect data via voice and screen recordings during participants' programming sessions. The data processing pipeline is shown in Figure 1.

Transcription. We transcribed the speech recordings using the automatic speech recognition tool WHISPER[6]. WHISPER recognizes text in multiple languages, even within the same sentence. This is useful since most participants spoke in German while using English for most technical terms. We pre-processed the transcriptions to facilitate subsequent annotation by adding line breaks at sentence ends and removing filler words (e.g., 'okay') that did not convey specific meaning. It is important to note that during the annotation process, the original recordings remain essential, as vocal tone can convey emotions and irony more effectively than written text.

Annotation. We developed a tool to annotate the transcript within the framework outlined in Section III. This tool enables us to manually label each line of the transcript (one line equals one sentence) and to assign it to exactly one utterance. We can also create episodes and assign utterances to them, editing episode attributes such as topic type (CODE, BUG, ...) and finish type (TRUST, LOST SIGHT, ...). In pair programming sessions, we recorded which programmer contributed to which lines, using labels 'Speaker A' and 'Speaker B' for anonymity. We annotated the transcripts manually from all 13 sessions, segmented them into utterances and episodes, and classified the utterances and episodes as described earlier.

Segmentation. We differentiate between monologues and dialogues, as human–human pair programming involves two people, while human–AI pair programming features a single person articulating their thoughts as a result of the think-aloud process. To segment the speech into utterances, we evaluate each spoken contribution determining whether it continues a previous utterance or begins a new one. This decision is based on two factors: whether the contribution logically relates to the preceding one (i.e., addressing the same topic) and whether a noticeable pause or interruption occurs. In dialogues, multiple consecutive utterances by one person can happen, but a new utterance begins when the other person speaks.

Classification. For each utterance, we manually assess its contribution to knowledge transfer. In most cases, utterances related to knowledge transfer contain an exchange of information between the two humans or between the human and GITHUB COPILOT, as verified through the voice recording. We consider utterances that repeat information from external sources, such as documentation or Web-search results, as part of knowledge transfer if it is apparent that it enhances a person's knowledge or understanding. If the audio-stream alone lacks context, screen recordings were consulted. From these utterances contributing to knowledge transfer, we identify those indicating a need for knowledge or indicating a start of sharing knowledge, marking the start of an episode (Definition 3).

The episode continues until an utterance indicating one of the five discussed outcomes (e.g., "Yes, this makes sense," classified as the finish type ASSIMILATION). While the audio transcript was sufficient in most cases, we referred to the

[6] https://openai.com/index/whisper/

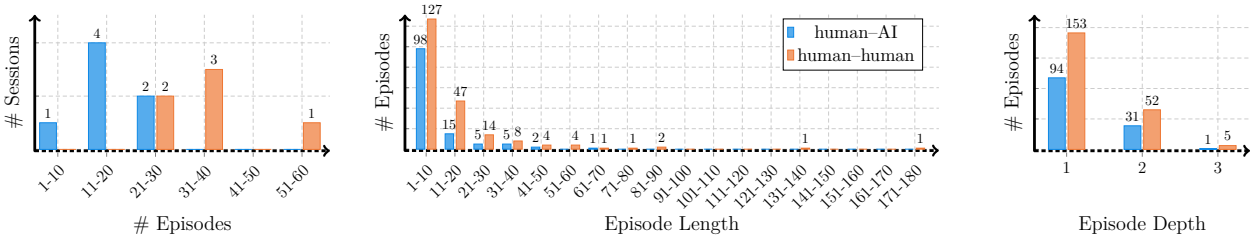

Figure 2: Comparison of episodes in terms of number (a), length (b), and depth (c).

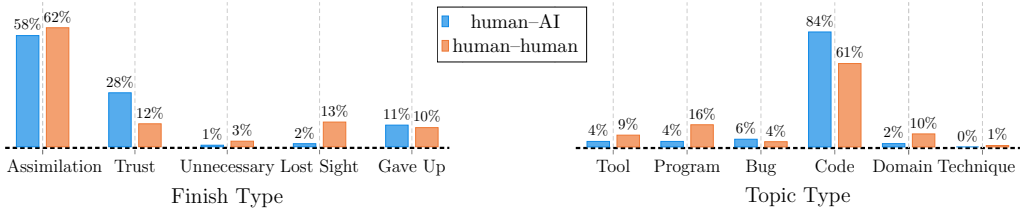

Figure 3: Distribution of finish types (a) and topic types (b).

original recordings when episode endings were unclear. If an episode covers multiple topics but eventually returns to the original one, a new nested episode is recorded.

Evaluation. We analyze the number of episodes, initiated by a need for knowledge, as well as the length and depth of episodes. Additionally, we use annotations and classifications as described above to compare human-human and human–AI pair programming settings. During classification, we also examine recurring session patterns, which are discussed further in Section VI. For all statistical tests, we set the significance level to $\alpha = 0.05$. While the applied statistical tests account for the given sample size, the number of participants (19 in total) limits a straightforward generalization of our findings. Thus, the reported quantitative results should be interpreted with appropriate caution. For the same reason, we refrain from reporting effect sizes, as these would not be reliable given the small sample. In addition to what follows, we provide a comprehensive list of the statistical tests and their results in our replication package.

V. Results

In this section, we evaluate and compare the knowledge transfer between traditional pair programming settings (human–human pair programming) and those involving a human paired with GitHub Copilot (human–AI pair programming) according to our two research questions.

A. Demographics

The following demographic characteristics provide essential context for our evaluation, serving as descriptive statistics for interpreting the study's findings. Of the 19 participants, 16 identified as male (84.2%) and 3 as female (15.8%).

All participants were pursuing a computer science bachelor's or master's degree. The average age of participants was 23.0 years, and participants rated their English proficiency at an average of 8.05 out of 10 points.

With a mean value of 6.21 out of 10, the self-indicated programming experience of our participants is rather high. Furthermore, there is a positive Spearman's rank correlation between the self-indicated absolute programming experience and the self-indicated experience relative to the participants' peers, $r_s(19) = 0.62$. These experience measures are based on the work of Siegmund et al. [29]. In addition to programming experience, participants reported a mean Python expertise of 5.53 out of 10, reflecting solid familiarity with the study's chosen language. However, their prior exposure to GitHub Copilot (2.47) and pair programming (4.40) was moderate. A Mann-Whitney-U test did not yield any statistically significant differences for any of the five experience questions between the two groups. Our sample reflects the characteristics of a typical computer science student population with some prior programming experience.

However, the difference in experience with GitHub Copilot stands out, with participants of the human–AI pair programming group rating their experience with GitHub Copilot, on average, at 4.43 (on a scale from 1 to 10), compared to an average rating of only 1.33 for the pair programming participants. As a reminder, these differences were due to chance and all participants only answered the questions after being randomly assigned to a group.

B. Characteristics of Knowledge Transfer Episodes

We begin by examining the differences in frequency, length and depth of knowledge transfer episodes guided by RQ1.

Frequency of Knowledge Transfer. As discussed in Section IV, not every sentence in a conversation contributes to knowledge transfer. Identifying significant differences in the number of episodes helps indicate how much and which parts of the conversation contribute to knowledge transfer. To this

end, we counted all episodes of both groups, regardless of their outcome or topic type.

In human–human pair programming sessions, there were in total 210 episodes, compared to 126 episodes in human–AI pair programming sessions. On average, pair programmers had 35.0 episodes per session and GitHub Copilot users had 18.0 episodes per session. Figure 2a contains the number of episodes per session across the two groups. The participants using GitHub Copilot encounter 15 to 22 episodes, with a median of 17. In contrast, pair programmers ran into 28 to 35 episodes, with a median of 33.5. A Welch's t-test reveals that the mean difference between the number of episodes of the two groups is statistically significant, $t(8.15) = -3.38$.

Length and Depth of Episodes. The lengths of episodes for both groups is shown in Figure 2b. On average, an episode of a human–human pair programming session spans 14.4 utterances, compared to 9.4 utterances in human–AI pair programming episodes. Notably, two extreme outliers in the human–human pair programming group could indicate that some episode endings may have been missed during annotation. However, a manual inspection did not confirm this. Instead, both outliers are top-level episodes that include smaller, nested episodes, thereby inflating their overall length. This is to be expected, as our definition of episode length includes all utterances within an episode, even if they belong to nested episodes. As a result, top-level episodes that encompass other episodes tend to be longer. The median is higher in the human–human pair programming group with 8 utterances compared to 6.5 utterances for the human–AI pair programming group. According to a Mann-Whitney U test, the distributions differ significantly. Figure 2c shows the distribution of episodes depths. The highest occurring depth is 3 for both groups, and their distributions appear to be similar, with a Mann-Whitney U test revealing no significant difference between the two groups. In both groups, most episodes are top-level episodes (i.e., a depth of 1). Thus, they were caused by a need for knowledge that did not result from another foregoing episode. Only one quarter of all episodes are stacked on top of a different one, and a negligible small share of 1–2 % of episodes is stacked on two episodes.

C. Characterizing Knowledge Transfer Dynamics

We continue with RQ2, that is, the question of differences in topic type and outcome of knowledge transfer episodes.

Content of Episodes. The distribution of episode topic types is shown in Figure 3b. The y-axis shows the percentage of episodes with the bars placed at the respective topic type on the x-axis. As the number of episodes is different for the two groups, we chose relative numbers for the plot. A χ^2-test reveals a significant difference between the two distributions.

Across both groups, the topic type Code occurs most frequently. However, it occurs significantly more in the human–AI pair programming group, accounting for 84 % of all episodes, with other topic types only occurring in at most 6 % of all episodes. In both groups, there was little discussion about abstract programming paradigms (with 0 % and 1 % of episodes having topic type Technique).

Figure 4: Episode with finish type Trust. Guillemets mark single utterances. Bracketed text added for context.

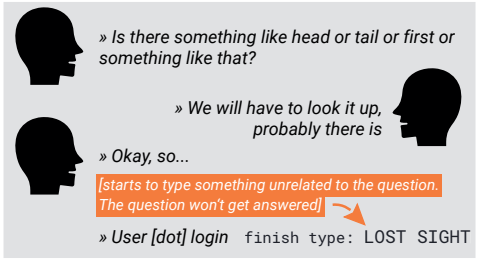

Figure 5: Episode with finish type Lost sight. Guillemets mark single utterances. Bracketed text added for context.

Outcomes of Episodes. We analyze the outcomes of knowledge transfer episodes with regard to finish types to assess the quality of knowledge transfer. This is the most crucial aspect because knowledge transfer is only useful to the customer if they can process and understand the information conveyed to them.

Figure 3a shows how finish types are distributed per group. On the y-axis, we see the percentage of episodes with the respective finish type. According to a χ^2-test, the distributions differ significantly. Both groups tend to finish most episodes successfully, with a 12% difference between the two settings in successful knowledge transfer episodes(finish type: Assimilation and Trust). More interesting is the discrepancy in the category Trust, with GitHub Copilot users achieving this finish type more than twice as often than users in human–human pair programming. Upon closer examination of episodes finish type Trust, we observe that in many GitHub Copilot sessions, programmers tend to accept the assistant's suggestions with minimal scrutiny, relying on the assumption that the code will perform as intended. Figure 4 shows an example of this happening. This issue does not come up in pair programming sessions so much, indicating that pair programmers rather pursue problems until the need for knowledge is satisfied.

Another category with a huge discrepancy is Lost sight. Figure 3a shows that this almost does not occur in the human–AI pair programming group, but one eighth of episodes of the human–human pair programming group are interrupted by an unrelated issue (see Figure 5 for an example).

By performing χ^2-tests to compare the occurrence frequen-

cies of individual finish types, we found significant differences between the two groups for the TRUST and LOST SIGHT finish types, while we could not find significant differences for the other three finish types.

VI. DISCUSSION

Even the most experienced programmer will reach a point in their day-to-day programming where some additional knowledge would be beneficial. In our case, the programming scenario involved a fairly self-contained programming task, during which we already observed plenty of knowledge transfer. How this knowledge transfer differs between traditional human pair programming and the AI-assisted approach was the core of our investigation that we will discuss in the following.

A. Findings

Regarding RQ1, our initial objective was to determine whether knowledge transfer occurs within human–AI pair programming and how the frequency, length, and depth of knowledge transfer episodes in human–AI pair programming compare to human–human pair programming.

We followed the definition by Zieris and Prechelt [6] and defined knowledge transfer as *any attempt* to close a perceived knowledge gap through the exchange or creation of new knowledge. These attempts to close knowledge gaps indeed occurred in both scenarios. The average number of episodes per session differs significantly between the groups, which may indicate either a greater need for knowledge exchange or simply a lower threshold for initiating these interactions in human–human pair programming. This suggests that individuals may be more eager to engage with each other.

In a manual investigation of the transcripts, we found many relatively short question-and-answer episodes in human–human pair programming sessions that do not emerge in human–AI pair programming sessions, supporting our assumption. These question-and-answer episodes usually follow a pattern in which the customer casually poses a question that the partner can answer quickly. This does not occur when using the GITHUB COPILOT assistant as, at the time the study was conducted, the only way to interact with the assistant was by waiting for a suggestion. Although it was technically possible to prompt the assistant through comments, this method was unintuitive, making it unlikely that participants would pose minor questions.

Finding 1: Knowledge transfer occurs in both human-human and human–AI pair programming settings, revealing knowledge gaps and efforts to bridge them. However, in human–human pair programming settings, individuals tend to engage more actively with each other, often involving relatively short question-and-answer episodes.

We examined the length of episodes by counting the number of utterances. The median episode length is significantly higher in the human–human pair programming group compared to the human–AI pair programming group. This difference might be expected, as a dialogue between two human partners naturally involves more exchanges than a monologue where one person thinks aloud. However, one might also assume that discussions in a pair programming setting would lead to faster conclusions–after all, that is one of the reasons for pair programming–but our study did not observe this effect. Without the proper context or goal in mind, it is difficult to conclude which method is superior. Shorter discussions might work well when quick decision-making is needed, while longer ones allow for thorough analysis and idea development.

Episodes with higher depths occur when the participant in the customer role requires knowledge on a topic distinct from the current episode's topic. In both groups, most episodes remain top-level. Deeper and longer episodes may suggest that certain explanations are unclear, as additional context often becomes necessary to clarify related topics. Interestingly, there is no significant difference in the distribution of episode depths.

Finding 2: While human–human pair programming interactions tend to produce longer episodes, the depth of episodes remains similar across both groups. This suggests that the need for additional contextual explanations is comparable.

Addressing RQ2, we analyzed how the quality and diversity of knowledge transfer episodes in pair programming with GITHUB COPILOT compare to human–human pair programming by examining the frequency of different topic types and finish types in both groups.

For both settings, most episodes were assigned the CODE topic type. During manual investigation, we found that this often included episodes in the human–AI pair programming setting where a programmer creates code with the help of the assistant and verifies (verbally) that the code actually fulfills its purpose. This form of knowledge transfer was named a "co-production" by Zieris and Prechelt [6]. Originally, for human–human pair programming, this defines episodes where the programmers work together on building new knowledge (e.g., discovering a bug and its cause) [6]. This again shows that in such scenarios, similar to the human–human pair programming setting, the programmer actively builds new knowledge by trying to understand the code presented by GITHUB COPILOT.

When comparing the topic types of knowledge transfer, we observe that there is a broader distribution in the human–human pair programming in which other topic types besides CODE occur more often than in the settings where GITHUB COPILOT is involved. We hypothesize that this is due to episodes with topic types such as PROGRAM or DOMAIN being more likely to start with casual conversation than a concrete question, making them more likely to occur with a human partner. Additionally, the suggestions made by GITHUB COPILOT are typically closely tied to the project's source code, making the emergence of unrelated topics unlikely.

Regarding less prevalent topic types, there are only few episodes on the topic type PROGRAM (i.e., thus, regarding syntax or semantic of the programming language). This can be explained by the high PYTHON experience that participants had, on average. Also, participants rarely discussed programming

concepts (topic type TECHNIQUE). They might not have needed to address them because the design of the project as well as the conditions under which they were supposed to work were all given. The relative rarity of the BUG topic type is likely because, when errors occur, discussions tend to focus less on their cause and more on finding a solution. In some cases, this also led to the code in question being deleted and rewritten, which meant that the reference to the error was usually removed.

Finding 3: In general, human–AI pair programming sessions focus more on the CODE topic type. Conversely, knowledge transfer in human–human pair programming involves a broader range of topic types, beyond mere code-related discussions.

Regarding the initiation of episodes, Zieris and Prechelt [22] differentiate between pull mode, where one person actively seeks information from their partner to close a knowledge gap, and push mode, where one person shares information based on the perception that their partner might need it. We cannot directly infer push mode episodes coming from GITHUB COPILOT, since all episodes in the human–AI pair programming group are purely comprised of utterances by a human participant. However, we noticed some interesting patterns, which show that GITHUB COPILOT can sometimes convey knowledge subtly, even without an explicit prompt or preceding comment. Most participants in the human–AI pair programming group did not have any experience with the database API SQLALCHEMY, which was used in the project. Hence, most forgot to add a commit statement after doing database transactions in the code. In many cases, GITHUB COPILOT suggested that adding such commit statement is crucial for the program to function correctly. Upon reading the suggestion in one function, participants realized that they were to add this statement in every other function that modifies the database as well. This elegantly shows that knowledge transfer does not exclusively happen "on request" but can be rather subtle and unsolicited. Programmers may learn something not only from another human but also via an AI coding assistant being proactively pushing suggestions.

Finding 4: GITHUB COPILOT can also convey critical information subtly. However, we suspect the transfer of subtle, tacit knowledge to occur less frequently in real-world settings where GITHUB COPILOT lacks access to company-specific information and domain expertise.

In human–AI pair programming, we observe a significantly higher share of successful knowledge transfer episodes (finish types ASSIMILATION and TRUST) than in human–human pair programming (see Figure 3). A key reason for this finding appears to be the significantly higher probability of distraction in the human–human setting (finish type LOST SIGHT) compared to human–AI pair programming.

In addition to the low likelihood of distraction, we find that knowledge transfer with the finish type TRUST occurs a lot more frequently in human–AI pair programming. Programmers tend to accept the assistant's suggestions with minimal scrutiny, assuming that the code will perform as intended.

Finding 5: Programmers tend to accept the assistant's suggestions with little critical review, often trusting that the code will work as expected—a tendency seen far more frequently than in the human–human setting. Conversely, conversations initiated with GITHUB COPILOT are less likely to be aborted due to programmers getting distracted.

B. Implications for Students and Practitioners

Beyond describing our findings, we now discuss their practical significance and outline implications for students and practitioners. We found a high level of TRUST episodes in human–AI pair programming sessions. If this pattern were to generalize beyond our setup, this would carry important real-world implications, warranting further investigation. These frequent TRUST episodes can reduce opportunities for deeper learning. For instance, students may accept suggestions without sufficient verification, leading to a more superficial understanding of the underlying concepts. For practitioners, unwarranted trust may leave important knowledge lacks unnoticed and lead to the introduction of bugs that can be hard to detect without deeper knowledge of the code base.

We also note that prior knowledge with tools like GITHUB COPILOT may influence how strongly such trust patterns emerge. In our sample, experience levels varied, with some participants reporting no prior use. While we cannot isolate its exact effect in this study, it is plausible that greater familiarity could help developers calibrate their trust, while inexperience may amplify over-reliance. As a consequence, students and practitioners should be made aware of the limitations of such AI tools and encouraged to critically assess their suggestions. Integrating AI tools into programming courses can enhance learning, but it requires careful guidance to prevent over-reliance, which is in line with prior findings [30, 31]. Proposed approaches should ensure that students actively learn and build essential critical thinking skills [32, 33].

Furthermore, in human–human pair programming sessions, the finish type LOST SIGHT occurred more often, that is, the episode was interrupted by an unrelated issue or topic. Our manual analysis suggests that this outcome largely arises because human partners often start side discussions that are unrelated to the original topic. For instance, each partner may prioritize a different topic in the discussion. By contrast, we did not find this to occur in human–AI pair programming, where the conversation is directed by the programmer and the AI simply responds to their input. In practice, the use of AI may increase immediate efficiency, as suggested by some studies [34], but reduce the broader knowledge exchange that arises from side discussions in human–human pair programming, potentially decreasing long-term efficiency. Thus, it might turn out that, AI is useful for simple, repetitive tasks where side discussions are less valuable, but when it comes to building deeper knowledge it must be treated with care, especially for students. While knowledge transfer appeared

comparable across conditions, human collaboration continues to offer unique value—particularly in areas such as design reasoning, mentoring, and contextual understanding—that go beyond what current AI systems can provide. Also, human–human pair programming collaboration can strengthen team communication, increase enjoyment of work, reduce staff-loss risk, and shorten onboarding time [35, 36, 8]. It is therefore advisable not to, at this point, consider fully automating the pair programming process, but to keep the advantages of both scenarios in mind and arrive at a mixed approach in practice.

C. Threats to Validity

Internal Validity. Think-aloud protocols may not capture all participant thoughts, as verbalization can be incomplete or cognitively demanding, potentially reducing performance or slowing task completion [37]. As a result, transcripts—especially in the human–AI pair programming group—may miss knowledge transfer episodes, suggesting our findings represent a lower bound. On the other hand, thinking aloud may also encourage structured thinking and improve task focus.

Additionally, data was annotated by a single person, introducing the risk of individual bias in the classification of knowledge transfer episodes. To mitigate this, we automated large parts of the processing and manually cross-checked a random sample of annotations with four additional reviewers.

Finally, while the sample size of 19 participants is, at least, in line with standards for similar mixed-methods research in the software engineering context, we cannot rule out that individual differences in the groups may have influenced the results. We mitigated this factor by randomly assigning participants to groups and by considering factors such as programming experience to check for group differences. The only statistically significant difference between the groups was in self-perceived experience with GitHub Copilot. While variation in the GitHub Copilot experience within the human–AI pair programming group might influence how individual participants engage with the tool, we do not consider this to threaten the validity of our findings, as only one of the two groups used GitHub Copilot, and our findings stem primarily from comparing interaction dynamics with and without the tool. That said, we acknowledge that different experience levels—especially within the GitHub Copilot group—might lead to variations in usage patterns.

External Validity. Participants had solid programming experience and reflected a typical gender distribution of computer science students (see Section V: Demographics) . To isolate the effect of human–AI pair programming on knowledge transfer, we controlled for factors like domain knowledge, peer relationships, and organizational culture. While this strengthens internal validity [38], it may limit ecological validity, as collaborative behavior can depend on personal traits and real-world dynamics. Nonetheless, our selection of students provides a meaningful starting point for studying knowledge transfer between humans and AI coding assistants, as knowledge transfer is very relevant to students given their ongoing learning. We therefore see this as a valuable first step toward understanding how developers interact with AI tools, while welcoming future studies with other populations to further broaden the evidence base.

The observed tendency to trust GitHub Copilot may result from situational factors like time pressure, lack of consequences for errors, or the perceived effort of manual verification.

We used Python for the programming task, which may influence GitHub Copilot 's performance [23]. However, we have no indication that our core findings on knowledge transfer would not generalize to other commonly used languages.

Construct Validity. Episode length was operationalized by utterance count, not time, due to missing timestamps. This approach does not reflect processing time, but emphasizes the amount of content exchanged, making it more robust against variation in speaking pace. Apart from that, we drew on existing frameworks to examine the relevant constructs.

VII. Conclusion

The promise of AI coding assistants evolving into true *AI pair programmers* has been a compelling narrative. Our findings show that, while they support knowledge transfer and offer proactive guidance, AI assistants do not yet replicate the diversity of human collaboration.

In our study with 19 participants, knowledge transfer occurred both in human–human pair programming and with GitHub Copilot. Notably, GitHub Copilot can subtly remind developers of important details, such as committing database changes, that might otherwise be overlooked. This unintentional form of knowledge transfer was previously assumed to be limited to human collaboration. Human–human pair programming enables spontaneous interactions but also increases the risk of distraction. In contrast, knowledge transfer with GitHub Copilot is less likely to be aborted, yet suggestions are often accepted with less scrutiny. This highlights the need for further research—technical and social—to understand trusting behavior and to develop mechanisms that encourage critical evaluation.

Our results suggest that combining AI assistants with traditional pair programming can balance efficiency with richer knowledge exchange. For instance, senior developers can offer contextual insights beyond AI capabilities, but may benefit from GitHub Copilot's support with repetitive tasks.

Data Availability

For transparency and reproducibility, we share all study artifacts online[7].

Acknowledgments

The authors thank the study participants for their time and participation. This work has been supported by the European Union as part of the ERC Advanced Grant Brains On Code (101052182).

[7] https://github.com/se-sic/knowledgeTransferCopilot

REFERENCES

[1] T. Dybå, E. Arisholm, D. I. Sjoberg, J. E. Hannay, and F. Shull, "Are two heads better than one? on the effectiveness of pair programming," *IEEE software*, vol. 24, no. 6, pp. 12–15, 2007.

[2] D. L. Jones and S. D. Fleming, "What use is a backseat driver? A qualitative investigation of pair programming," in *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, C. Kelleher, M. M. Burnett, and S. Sauer, Eds. IEEE Computer Society, 2013, pp. 103–110.

[3] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. W. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, I. Babuschkin, S. A. Balaji, S. Jain, A. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *ArXiv*, vol. abs/2107.03374, 2021.

[4] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Chi conference on human factors in computing systems extended abstracts*, 2022, pp. 1–7.

[5] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Measuring github copilot's impact on productivity," *Communications of the ACM*, vol. 67, no. 3, pp. 54–63, 2024.

[6] F. Zieris and L. Prechelt, "On knowledge transfer skill in pair programming," in *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM*, M. Morisio, T. Dybå, and M. Torchiano, Eds. ACM, 2014, pp. 11:1–11:10.

[7] S. K. Kuttal, B. Ong, K. Kwasny, and P. Robe, "Trade-offs for substituting a human with an agent in a pair programming context: The good, the bad, and the ugly," in *CHI '21: CHI Conference on Human Factors in Computing Systems*, Y. Kitamura, A. Quigley, K. Isbister, T. Igarashi, P. Bjørn, and S. M. Drucker, Eds., 2021, pp. 243:1–243:20.

[8] L. Plonka, H. Sharp, J. Van der Linden, and Y. Dittrich, "Knowledge transfer in pair programming: An in-depth analysis," *International journal of human-computer studies*, vol. 73, pp. 66–78, 2015.

[9] M. Stigmar, "Peer-to-peer teaching in higher education: A critical literature review," *Mentoring & Tutoring: partnership in learning*, vol. 24, no. 2, pp. 124–136, 2016.

[10] J. Hattie, *Visible learning: A synthesis of over 800 meta-analyses relating to achievement*. routledge, 2009.

[11] S. Imai, "Is github copilot a substitute for human pair-programming? an empirical study," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 319–321.

[12] S. O. Bada and S. Olusegun, "Constructivism learning theory: A paradigm for teaching and learning," *Journal of Research & Method in Education*, vol. 5, no. 6, pp. 66–70, 2015.

[13] T. J. Nokes-Malach, J. E. Richey, and S. Gadgil, "When is it better to learn together? insights from research on collaborative learning," *Educational Psychology Review*, vol. 27, pp. 645–656, 2015.

[14] D. W. Johnson, R. T. Johnson, and K. Smith, "The state of cooperative learning in postsecondary and professional settings," *Educational psychology review*, vol. 19, pp. 15–29, 2007.

[15] B. Goldschmid and M. L. Goldschmid, "Peer teaching in higher education: A review," *Higher education*, vol. 5, no. 1, pp. 9–33, 1976.

[16] C. H. Liu and R. Matthews, "Vygotsky's philosophy: Constructivism and its criticisms examined." *International education journal*, vol. 6, no. 3, pp. 386–399, 2005.

[17] J. Piaget, "The psychogenesis of knowledge and its epistemological significance," in *Language and Learning: The Debate Between Jean Piaget and Noam Chomsky*, M. Piattelli-Palmarini, Ed. Harvard University Press, 1980, pp. 1–23.

[18] K. Beck, "Embracing change with extreme programming," *Computer*, vol. 32, no. 10, pp. 70–77, 1999.

[19] E. Arisholm, H. Gallis, T. Dybå, and D. I. K. Sjøberg, "Evaluating pair programming with respect to system complexity and programmer expertise," *IEEE Trans. Software Eng.*, vol. 33, no. 2, pp. 65–86, 2007.

[20] A. Begel and N. Nagappan, "Pair programming: what's in it for me?" in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, 2008, pp. 120–128.

[21] F. Zieris and L. Prechelt, "Observations on knowledge transfer of professional software developers during pair programming," in *Proceedings of the 38th International Conference on Software Engineering, ICSE*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 242–250.

[22] F. Zieris, "Qualitative analysis of knowledge transfer in pair programming," Ph.D. dissertation, 2020.

[23] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *19th IEEE/ACM International Conference on Mining Software Repositories, MSR*. ACM, 2022, pp. 1–5.

[24] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of ai on developer productivity: Evidence from github copilot," 2023. [Online]. Available: https://arxiv.org/abs/2302.06590

[25] Q. Ma, T. Wu, and K. Koedinger, "Is AI the better programming partner? Human-Human Pair Programming vs. Human-AI pAIr Programming," 2023. [Online]. Available: https://arxiv.org/abs/2306.05153

[26] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 85–111, 2023.

[27] V. Stray, N. B. Moe, N. Ganeshan, and S. Kobbenes, "Generative ai and developer workflows: How github copilot and chatgpt influence solo and pair programming," 2025.

[28] G. Fan, D. Liu, R. Zhang, and L. Pan, "The impact of ai-assisted pair programming on student motivation, programming anxiety, collaborative learning, and programming performance: a comparative study with traditional pair programming and individual approaches," *International Journal of STEM Education*, vol. 12, no. 1, p. 16, 2025.

[29] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg, "Measuring and modeling programming experience," *Empirical Software Engineering*, vol. 19, pp. 1299–1334, 2014.

[30] M. Lepp and J. Kaimre, "Does generative ai help in learning programming: Students' perceptions, reported use and relation to performance," *Computers in Human Behavior Reports*, vol. 18, p. 100642, 2025.

[31] A. Park and T. Kim, "Code suggestions and explanations in programming learning: Use of chatgpt and performance," *The International Journal of Management Education*, vol. 23, no. 2, p. 101119, 2025.

[32] B. A. Becker, P. Denny, J. Finnie-Ansley, A. Luxton-Reilly, J. Prather, and E. A. Santos, "Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 500–506.

[33] R. Choudhuri, A. Ramakrishnan, A. Chatterjee, B. Trinkenreich, I. Steinmacher, M. Gerosa, and A. Sarma, "Insights from the frontline: Genai utilization among software engineering students," in *2025 IEEE/ACM 37th International Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, 2025, pp. 1–12.

[34] M. I. H. Shihab, C. Hundhausen, A. Tariq, S. Haque, Y. Qiao, and B. W. Mulanda, "The effects of github copilot on computing students' programming effectiveness, efficiency, and processes in brownfield coding tasks," in *Proceedings of the 2025 ACM Conference on International Computing Education Research V. 1*, 2025, pp. 407–420.

[35] J. Vanhanen and C. L. Lassenius, "Perceived effects of pair programming in an industrial context," in *33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007)*, 2007, pp. 211–218.

[36] A. Cockburn and L. Williams, "The costs and benefits of pair programming," 02 2000.

[37] E. Charters, "The use of think-aloud methods in qualitative research an introduction to think-aloud methods," *Brock Education Journal*, vol. 12, no. 2, 2003.

[38] J. Siegmund, N. Siegmund, and S. Apel, "Views on internal and external validity in empirical software engineering," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 9–19.